

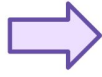


Assembly & Registers



The **compiler** converts our **C code** into low-level **assembly instructions**, which can then be converted into machine code (0s and 1s) run by your computer.

```
int gold = 10;
int treasure = 5;
gold += treasure;
```

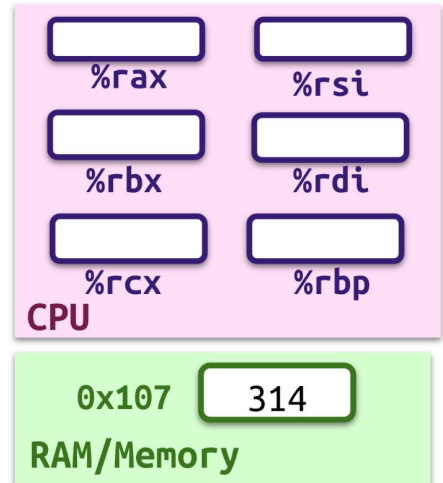


```
mov $10, %rax
mov $5, %rbx
addl %rbx, %rax
```

Think in terms of **registers**: variable boxes used as temporary storage / **scratch paper** for programs.

A register is an **very fast read/write slot** right on the CPU (not in memory), which can hold variable values

Move data in & out of registers to perform operations on it with the CPU, e.g., arithmetic operations, parameters into functions, return values



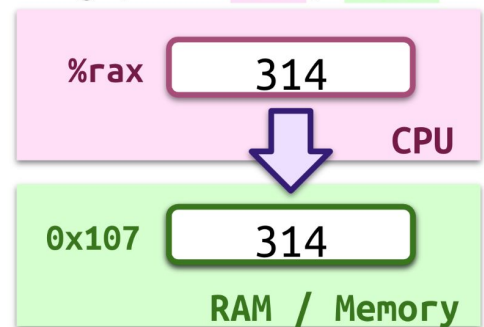
MOV Instruction



Key Instruction: mov src, dst

- Responsible for **copying data** around!
- src** can be an immediate (constant value), register, or memory location
- dst** can be a register or memory location
- Exception: src and dst cannot both be memory locations!** (must first temporarily copy the data into a register)

e.g., `mov %rax, 0x107`



Instructions like `mov` can **specify via a suffix the # of bytes** being moved, etc.

- `movb` for 1 byte being copied
- `movl` for "double word" (4 bytes)
- `movw` for "word" (2 bytes)
- `movq` for "quad word" (8 bytes)

Addressing Modes in Assembly



Immediate: `mov $0x5, dst`

A **dollar sign (\$)** indicates source is **constant value** to be moved into dst

Direct: `mov 0x106b, dst`

Fixed, static memory address

This reads from **Mem[0x106b]**, the memory at address 0x106b.

Indirect with Displacement:
`mov 8(%rax), dst`

Same as indirect, but has a **number prefix as bytes plus the address**, like pointer arithmetic!

The above reads from **Mem[\$rax + 8]**

Register: `mov $rax, dst`

Moves **value held in register** to dst.

Indirect: `mov ($rax), dst`

This reads from **Mem[\$rax]**, where the register \$rax holds an address, e.g., 0x106b

Note **parentheses** to contrast with above.

Indirect with Displacement + Scaled Index: `mov 8($rsp, $rcx, 4), dst`

More complex displacement: **offset + scaling factor** (e.g., 1,2,4,8), often to index into arrays.

Reads from **Mem[\$rsp + 8 + (4 * \$rcx)]**.



lea Instruction



Key Instruction: **lea** src, dest

- Similar to mov, **except that it doesn't dereference / look at memory**, i.e. it copies of the value of src itself to a register, not the data at the address src.
- It's really just pointer arithmetic! 🤔

%rax 0x100
%rcx 0x10
%rdx CPU

2002 2025
 0x100 0x140
 RAM/Memory

leaq (%rax, %rcx, 4), %rdx

%rdx 0x140 ← holds **%rax + 4 * %rcx**

lea doesn't dereference this address/look at memory!

movq (%rax, %rcx, 4), %rdx

%rdx 2025 ← holds **Mem[%rax + 4 * %rcx]**

Meanwhile, **mov** dereferences the address to get 2025!

Registers: Great Power = Great Responsibility

Many registers are given designated roles or responsibilities, e.g., for function calls:

- The first 4 arguments passed in are **\$rdi, \$rsi, \$rdx, \$rcx**, respectively
- The return value of a function (if any) is written to **\$rax** before it ends

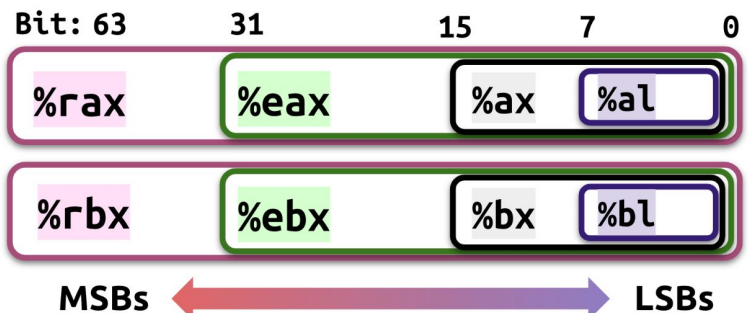


Registers Sizes & Subregisters

Registers can be **partially or fully referred to**. Example: we can access just **32 bits** of a register (e.g., **\$edi**) instead of all 64 bits (e.g., **\$rdi**), if we're operating on an **int**.

Specifically, a reference to the **e-named register** accesses the **lower 32-bit subregister**

%edi is subregister of **%rdi**
%eax is subregister of **%rax**



New GDB Commands for Assembly!



disassemble: prints the assembly for the currently executing function (can write *(gdb) disassemble func_name* to disassemble another function)

b *address: Places a breakpoint on a specific assembly instruction, via its address

stepi, nexti: Assembly versions of step, next, going one line of assembly at a time

info reg: Prints out the values in all integer registers

print \$register: Prints out a specific register's contents, e.g., **print \$rax**

Note: Can print using a typecast, e.g., *(gdb) print (char*) ptr*, or in hex, via *print /x*

*(gdb) p (char *)\$rax*

\$3 = 0x400720 "Hello, world!\n"

*(gdb) p *(long *)\$rax*

*(gdb) p *(long *)((char *)\$rsp + 8)*

set \$register = value: manually set a register's contents, useful for testing behavior

In Terminal / myth, can run **objdump -d executable_name** to list all assembly instructions